A New Routing Scheme for Jellyfish and its Performance with HPC Workloads

Xin Yuan
Dept. of Computer Science
Florida State University
Tallahassee, Fl 32312
xyuan@cs.fsu.edu

Santosh Mahapatra
Dept. of Computer Science
Florida State University
Tallahassee, Fl 32312
mahapatr@cs.fsu.edu

Wickus Nienaber
Dept. of Computer Science
Florida State University
Tallahassee, Fl 32312
nienaber@cs.fsu.edu

Scott Pakin
Los Alamos National Laboratory
Los Alamos, New Mexico
pakin@lanl.gov

Michael Lang
Los Alamos National Laboratory
Los Alamos, New Mexico
mlang@lanl.gov

ABSTRACT

The jellyfish topology where switches are connected using a random graph has recently been proposed for large scale data-center networks. It has been shown to offer higher bisection bandwidth and better permutation throughput than the corresponding fat-tree topology with a similar cost. In this work, we propose a new routing scheme for jellyfish that out-performs existing schemes by more effectively exploiting the path diversity, and comprehensively compare the performance of jellyfish and fat-tree topologies with HPC workloads. The results indicate that both jellyfish and fat-tree topologies offer comparable high performance for HPC workloads on systems that can be realized by 3-level fat-trees using the current technology and the corresponding jellyfish topologies with similar costs. Fat-trees are more effective for smaller systems while jellyfish is more scalable.

Categories and Subject Descriptors

C.2.1 [**Computer Systems Organization**]: Computer-Communication Networks—*Network Architecture and Design*

General Terms

Performance

Keywords

Interconnects, jellyfish, fat-tree, HPC workload, performance

1. INTRODUCTION

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
SC13 November 17-21, 2013, Denver, CO, USA
Copyright 2013 ACM 978-1-4503-2378-9/13/11 ...$15.00
http://dx.doi.org/10.1145/2503210.2503229.

The jellyfish topology where the switches are connected by a random graph has recently been proposed for large scale data center networks [14]. It has been shown that some jellyfish topologies offer larger bisection bandwidth and higher permutation throughput than the corresponding fat-trees with comparable costs [14]. Since fat-trees are the dominating topologies used in commodity HPC clusters, jellyfish is potentially an effective alternate topology for HPC clusters. Hence, it is necessary to understand the relative strengths of jellyfish and fat-tree topologies.

Although jellyfish provides high connectivity among switches, routing on jellyfish is a challenge. Singla, et al. [14] indicate that traditional shortest path routing and equal-cost multiple path (ECMP) routing [8] are not effective for the jellyfish topology, and proposes k-shortest path (KSP) routing. However, the jellyfish topology has several features that make the plain KSP ineffective. First, in the jellyfish topology, many processing nodes are connected to each switch; and the number of source-destination (SD) pairs from one switch to another is relatively large. Using KSP, all SD pairs from one switch to another will share the k-shortest paths between the switches. Hence, unless k is a large number, KSP may not be able to fully exploit the path diversity of the jellyfish topology. Second, since the switch level topology in jellyfish is a random graph, the number of "short" paths between each pair of switches is random. KSP cannot effective exploit the different numbers of short paths between different pairs of switches in the random topology.

We propose a new jellyfish routing scheme called Limited Length Spread k-shortest Path Routing (LLSKR) that overcomes the limitations of KSP. The main idea of LLSKR is to (1) allow different SD pairs from a source switch to a destination switch to use different sets of paths between switches (Spread k-path), and (2) only use short paths for communications (Limited Length). While each SD pair only uses a small number of paths with LLSKR, SD pairs from one switch to another collectively use a much larger number of paths between switches than with KSP. Moreover, LLSKR adapts to the different number of paths between different pairs of switches in jellyfish and uses different numbers of short paths between each pair of switches to carry traffic. As a result, LLSKR exploits the path diversity in jellyfish

more effectively than KSP by utilizing a much larger number of short paths while avoiding long paths. The performance study indicates that LLSKR consistently out-performs KSP for different network configurations and traffic patterns. In particular, when k is a relative small number such as 1, 2 and 4, LLSKR has much better performance than KSP, with an average improvement of up to 30% for a very large class of HPC traffic patterns. Note that in practical multi-path routing, the number of paths that can be used for a SD pair is often limited [15].

To obtain a good understanding of the strengths and limitations of jellyfish and fat-tree topologies for HPC platforms, we carry out a comprehensive comparative study of jellyfish and fat-tree using traffic patterns in HPC systems. One of the key questions this study seeks to answer is the following: from a performance perspective, is it worth connecting an HPC network randomly to exploit small-world network properties? The HPC workloads reflect a wide range of HPC applications as well as how HPC systems are used. State-of-the-art multi-path routing schemes for both jellyfish and fat-tree are considered in the study. The results indicate that for HPC workloads, jellyfish and fat-tree both offer comparable high performance on systems that can be realized with 3-level fat-trees using the current technology and the corresponding jellyfish topologies with similar costs. For such systems, with a common node allocation and mapping scheme, fat-trees support the common stencil communication patterns more effectively while jellyfish achieves higher performance for traffic with more global communications. In addition, fat-trees are more effective for smaller systems while jellyfish is more scalable.

The rest of the paper is organized as follows. We briefly introduce jellyfish and fat-tree topologies in Section 2. Section 3 describes the HPC workloads as well as the performance metric for evaluating the topologies. Section 4 presents the new routing algorithm for jellyfish and studies the performance of the new routing scheme. Section 5 presents our evaluation of jellyfish and fat-tree using HPC workloads. Finally, Section 6 concludes the paper.

2. BACKGROUND

2.1 Jellyfish topology

In the Jellyfish topology [14], the switches are connected using a random graph. We consider the case when all switches have same number of ports and servers (processing nodes): each switch has x ports, of which y ports are connected to other switches and $x - y$ ports are connected to processing nodes. Let N be the number of switches, the network supports $N(x - y)$ processing nodes. The network is a *random regular graph*, denoted as $RRG(N, x, y)$. A particular topology is uniform-randomly sampled from the sample space of all such topologies. Theoretical analysis of random graphs indicates that such a topology is efficient [2, 3]: almost every RRG has low diameter and high bisection bandwidth. Moreover, even though RRG's are random and adversary topologies such as disconnected graphs are special RRG's. In practice, the probability that such an adversary being generated is negligibly small. Most RRG's have very similar topology properties. For example, it is proven that most of RRG's with the same parameters have the same diameter [2].

Figure 1 shows an example of the jellyfish topology (RRG(5, 7, 3)). Note that as discussed in [14], a jellyfish topology may have one switch with one open port as the switch $sw3$ in Figure 1. Multiple ($x - y = 4$) processing nodes are connected to each switch. There are thus, $(x - y)^2$ SD pairs from one switch to another switch. In Figure 1, $(x - y)^2 = 16$ SD pairs from switch $sw0$ to switch $sw4$: $(s0, d0)$, $(s0, d1)$, $(s0, d2)$, $(s0, d3)$, ... $(s3, d0)$, $(s3, d1)$, $(s3, d2)$, $(s3, d3)$. Using shortest path routing, the shortest path between $sw0$ and $sw4$, which is $sw0 \rightarrow sw4$, will be used to carry all traffic for the 16 SD pairs. Hence, the shortest path routing algorithm is not effective: only one path between the pair of switches will be used to carry traffic for the $(x-y)^2$ SD pairs. In addition, since the switch topology is random, there are not many equal length shortest paths between a pair of switches, which renders Equal-cost Multi-path (ECMP) routing ineffective. Singla suggests k-shortest path (KSP) routing for jellyfish [14], where k shortest paths for each SD pair are used to carry traffic for the SD pair. However, due to the structure of the jellyfish topology, the k shortest paths between the source switch and the destination switch are shared by all SD pairs from the source switch to the destination switch. Hence, KSP only alleviates the problem with shortest path routing or ECMP to a degree: only k paths between each pair of switches are used to carry traffic for all SD pairs on the pair of switches: the k paths are shared by the $(x - y)^2$ SD pairs. Unless k is a large number, KSP can only exploit the path diversity to an extent. The plain KSP also cannot effectively exploit the different numbers of short paths between different pairs of switches in the random topology. When the number of short paths between a pair of switches is smaller than k, k-shortest path routing will result in longer paths being used, which may degrade the routing performance. On the other hand, when the number of short paths between a pair of switches is larger than k, some of the short paths will not be utilized, which may also affect the routing performance. Our proposed routing scheme overcomes these limitations.

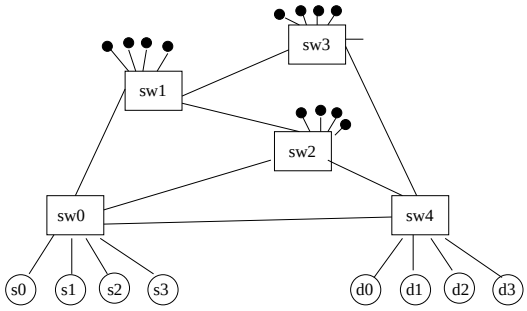

Figure 1: k-shortest path routing on jellyfish

There are practical issues for deploying the jellyfish topology. The cabling for a large-scale random topology can be a challenge. Random connections may require more long cables than a regular topology. Maintaining the random regular topology is also an issue since given choices, local (short) connections are in general more preferred than global (long) connections. It is also difficult to verify correct connectivity and identify faults on a random topology than a regular topology. This paper does not consider these practical issues. Our focus is on the performance of the topology with HPC workloads.

2.2 Fat-tree

The fat-tree topology has been widely adopted in HPC clusters and data centers due to its properties such as logarithmic diameter, scalability of bisection-bandwidth, and multiple shortest paths between any pair of processing nodes. We consider fat-trees built from the same switches and use the extended generalized fat-tree, $XGFT$ [12], to describe fat-trees in this paper. An extended generalized fat-tree $XGFT(h; m_1, ..., m_h; w_1, ..., w_h)$ has $h+1$ levels of nodes with nodes at level 0 only having parents, nodes at level h only having children, all other nodes having both parents and children. Each level i node, $0 \leq i \leq h-1$, has w_{i+1} parents; and each level i node, $1 \leq i \leq h$, has m_i children. The details for constructing an XGFT can be found in [12]. Like in [14], we only consider full bisection bandwidth fat-trees where $w_{i+1} : m_i = 1 : 1$ for all levels. Level 0 nodes are processing nodes while nodes in other levels are switches or routers. $XGFT(h; m_1, m_2, ..., m_h; w_1, w_2, ..., w_h)$ has $\prod_{i=1}^{h} m_i$ processing nodes (level 0 nodes); $\prod_{i=1}^{h} w_i$ top level switches (level h nodes). In general, $XGFT(h; m_1, m_2, ..., m_h; w_1, w_2, ..., w_h)$ has $(\prod_{i=k+1}^{h} m_i) \times (\prod_{i=1}^{k} w_i)$ switches (or processing nodes) at level k, $0 \leq k \leq h$.

Many routing algorithms have been developed for fat-trees. For single path routing, the most notable routing is the destination-mod-k routing that is widely used in InfiniBand networks. For multi-path routing, since there are many equal length shortest paths in a fat-tree, ECMP is effective for fat-tree. An earlier study has shown that when the number of paths used for each SD pair is limited, the path selection method for ECMP can have an impact on the routing performance [11]. In particular, using link disjoint paths (disjoint path heuristic [11]) for each SD pair results in higher performance than other path selection schemes such as random paths. In this study, we assume the disjoint path heuristic for multi-path routing on fat-trees.

3. HPC WORKLOADS AND PERFORMANCE METRIC

We first describe HPC traffic workloads used in the performance study. The workload depends both on the logical traffic patterns in HPC applications and on how the processing nodes are allocated and mapped to each application. We will then describe the performance metric used to evaluate the topologies.

3.1 HPC application traffic patterns

Table 1 summarizes the HPC communication patterns used in this study, which represent the patterns in a vast majority of HPC applications. The patterns include three types: (1) patterns that have traditionally been used for evaluating the performance of interconnects in HPC systems, (2) patterns in common HPC applications, and (3) primitive communication patterns that are sub-patterns of many common communication patterns. The first type includes the all-to-all pattern and bisect patterns. The performance of all-to-all communication closely correlates to the bisection bandwidth of a topology. In a bisect pattern, half of the processing nodes communicate with the other half of the processing nodes. The average performance of the bisect patterns is the effective bisection bandwidth [7].

The second type includes common patterns in HPC applications. Studies have indicated that the vast majority of HPC applications that run at scale have low-dimension stencil patterns [9, 13], including 2-dimension nearest neighbor (2DNN), 2-dimension nearest neighbor with diagonals (2DNNDIAG), 3-dimension nearest neighbor (3DNN) and 3-dimension nearest neighbor with diagonals (3DNNDIAG). These four types of traffic patterns are included in our evaluation. Another important class of HPC applications exhibit irregular or unknown communication patterns. Such applications include sparse linear system solvers and adaptive mesh refinement methods. For these applications, although the communication pattern is irregular, the number of destinations from each node is small relative to all nodes in the system. We include two traffic patterns for this type of applications: RANDN50 and RANDOM50. RANDOM50 are patterns where each SD pair has an equal probability to be included in a pattern and the total number of SD pairs is uniformly distributed from 1 to $nprocs \times 50$, where $nprocs$ is the number of processing nodes in the system. RANDN50 are patterns where all processing nodes have the same random number of random peers; the number of peers is randomly distributed from 1 to 50; and each node has an equal probability to be a peer.

The third type includes permutation patterns and shift patterns. Permutations are patterns where each node can send and receive at most one message. Any communication patterns can be partitioned into a sequence of permutations. In a shift pattern, each node i communicates with node $(i + a)$ mod $nprocs$, where $nprocs$ is the number of processing nodes and a can be any value from 1 to $nprocs$. Many point-to-point patterns such as 2DNN and collective communications such as all-to-all and all-gather operations can be decomposed into a set of shift or permutation patterns.

3.2 Process to node mapping

The physical communication pattern in an interconnect depends on not only the logical traffic patterns in HPC applications, but also node allocation, node numbering, and process to node mapping. Node numbering and process to node mapping for most production systems are proprietary. In this paper, we assume that for fat-trees, nodes are numbered such that nodes connected to one switch are numbered consecutively, and that nodes within a sub-fat-tree are numbered close to one another. This is a node numbering used in a production cluster Mustang with fat-tree interconnect at Los Alamos National Lab (LANL). For jellyfish topologies, we assume that switches are first ordered, and that the processing nodes attached to each switch are numbered consecutively. Using these node numbering schemes, we consider the following three node allocation and process to node mapping schemes.

- *Whole system direct mapping.* In this case, it is assumed that the whole system runs one application with one communication pattern. The application processes are mapped to nodes using the identity function.
- *Whole system random mapping.* The whole system runs one application with one communication pattern. Each application process is mapped to a random processing node with equal probability.
- *Job allocation trace-based allocation and mapping.* In

pattern	description
all-to-all	throughput for all-to-all pattern, correlate to bisection bandwidth
bisect	average throughput all bisect patterns, same as effective bisection bandwidth
2DNN	average throughput for all 2-dimensional nearest neighbor patterns with random 2D grid sizes
2DNNDIAG	average throughput for all 2-dimensional nearest neightbor with diagonals patterns with random 2D grid sizes
3DNN	average throughput for all 3-dimension nearest neighbor patterns with random 3D grid sizes
3DNNDIAG	average throughput for all 3-dimension nearest neighbor patterns with random 3D grid sizes
RANDOM50	average throughput for random patterns where the number of SD pairs is uniformly distributed between 1 to $nprocs \times 50$
RANDN50	average throughput for random patterns with the same number (uniformly distributed from 1 to 50) of random destinations for each node
permutation	average throughput for all permutation patterns
shift	average throughput for all possible shift patterns

Table 1: HPC communication patterns used in the evaluation

this case, we collected job allocation traces on a production 8894-node supercomputers *Cielo* at LANL. The job allocation on the supercomputers is sampled every 4 hours for about 7 weeks. Each trace includes the nodes allocated to each job. Since the sizes of the production clusters are different from the sizes of the systems that we study, we extrapolate the job allocation from production systems to the sizes of the target system being investigated. We assume that only the largest job is performing communication with one of the HPC patterns while the rest jobs are idle.

In general, with whole system direct mapping, application traffic locality can be preserved; with whole system random mapping, all communications are global communications. In practice, an application running on a suppercomputer usually uses multiple continuous partitions: the traffic pattern is a mix of direct mapping and random mapping. The job allocation traced-based allocation and mapping tries to capture the characteristics.

3.3 Performance metric

We use the average theoretical throughput for each type of traffic patterns as the performance metric to evaluate the performance of the network for each type of traffic patterns. The theoretical throughput for a given communication pattern is the aggregate throughput that can be achieved for the traffic pattern assuming (1) that the network has perfect congestion control and flow control, (2) that each flow in the pattern is communicated at its maximum possible rate, and (3) that the message size in the pattern is infinite. This metric reflects the application level communication performance. In the following, we will describe how the theoretical throughput for a given traffic pattern is computed, and how the average theoretical throughput for a class of traffic patterns is estimated.

The theoretical throughput depends on the routing algorithm. We consider two types of routing schemes on jellyfish and fat-tree: static single path routing where each flow follows a fixed path and multi-path routing with a Multi-path TCP (MPTCP) like protocol [6] where each flow is realized by multiple (TCP) sub-flows.

Given a communication pattern that consists of a set of SD pairs, for static single path routing, we compute the theoretical throughput for each flow using a similar approach as that in [7]. The throughput is derived based on the maximum link load. We assume that all links connecting processing nodes to the network have the same bandwidth (B), and that links between switches may have different bandwidths. Let inter-switch link have a bandwidth of b, we define the capacity of the link to be $\frac{b}{B}$, which denotes the number of flows that the link can support without being overloaded. All links connecting to processing nodes have a capacity of 1. For a static single path routing and given a pattern, we can compute the number of times that each link is used by the pattern. Let a link l with capacity C and used X times in a pattern, the link load is defined as $load_l = \frac{X}{C}$. For each SD pair (s, d) in the pattern, the throughput for the SD pair is the reciprocal of the maximum load in the link along the path:

$$T(s,d) = \frac{1}{\max_{l \in path(s,d)} load_l}.$$

By definition, the throughput is always a fraction between 0 and 1. The throughput for a flow can be interpreted as the fraction of the input bandwidth (maximum host injection rate) that the flow can use for the pattern. The throughput of the pattern is the average throughput of all flows in the pattern.

Using multi-path routing with an MPTCP like protocol, each flow is partitioned into several sub-flows and the system adaptively allocates data to each sub-flow depending on the rates for the sub-flows. Hence, multi-path routing with an MPTCP like protocol is theoretically able to exploit the maximum throughput in each path (sub-flow). This scheme can be modeled as the single path routing scheme with each sub-flows using a separate flow, and the rate for each sub-flow can be computed using the method for static single path routing. The rate for each flow in the pattern is the sum of the throughput of all of its sub-flows; and the aggregate throughput for a pattern is the sum of the throughput for all flows in the pattern.

For most of the traffic classes that we consider such as 2DNN, there are many instances. For example, there are many 2DNN patterns with different 2D grid sizes. The average throughput for a class of patterns is the average throughput across different patterns in that class. The number of patterns in each class can be very large; and it may be impossible to compute the exact average throughput for the class. We approximate the average throughput for each class by random sampling. For example, for random 2DNN patterns, the average throughput can be obtained by randomly generating a large number random 2DNN patterns and computing the average throughput for the random patterns. In this work, we use a statistical method to reduce the number of random samples needed. Specifically, we first generate 36 random samples, and compute the 95% confi-

dence interval. If the confidence interval is within 2% of the average, we consider the the average sufficient accurate and report the result. If the confidence interval does not meet the threshold, we double the number of samples and repeat the process until the threshold is reached or the number of samples is larger than 5000. For job allocation trace-based mapping, the average is over random patterns on randomly selected mapping from the trace.

By selecting a representative subset of traffic patterns that reflect a specific workload, the performance metric can be used to create site-custom performance comparisons. In this work, we use the set of communication patterns that in aggregate covers a generic HPC workload to give an overall performance comparison.

4. A NEW ROUTING ALGORITHM FOR JELLYFISH

We will use the notion *source switch* and *destination switch* of a SD pair to denote the switch directly connected to the source and the destination in the SD pair, respectively. *SD pairs on a pair of source and destination switches* are the SD pair whose sources are in the source switch and whose destinations are in the destination switch.

Limited Length Spread k-shortest path routing (LLSKR) overcomes the limitations of KSP by taking the unique topological features of jellyfish into consideration. First, since the SD pairs from one switch to another is relatively large ($(x-y)^2$) in the jellyfish topology, LLSKR incorporates a *spread* mechanism that allows different SD pairs on the same source and destination switches use different sets of paths, which spreads the traffic for the SD pairs over a much larger number of paths between the switches than KSP. Let the number of paths used for each SD pair be k, the largest number of paths between each pair of switches that can be used in LLSKR is $k(x-y)^2$ when each SD pair uses a different set of k paths. Second, the switch level topology in jellyfish is a random regular graph $RRG(N,x,y)$. The construct of $RRG(N,x,y)$ dictates what can be considered as a **short** path between the source and destination switches as will be discussed later. In addition, the number of short paths between each pair switches is random. LLSKR utilizes these properties and makes routing choices for each pair of switches based on the number of short paths between them.

In $RRG(N,x,y)$, each switch has y links connecting to other (random) switches. Hence, the total number of 1-hop paths from a source switch is y; the total number of 2-hop paths from the switch is roughly $y(y-1)$ (ignoring the paths with loops, each 1-hop switch has $y-1$ ports to reach 2-hop switches); and the total number of i-hop paths is roughly $y(y-1)^{i-1}$. Here, we ignore the small probability that the paths may have loops (repeated switches in the path). When N is large and i is small, the probability is very small. Since the topology is a random graph, the destinations of the paths are uniformly distributed among the N switches in the system. Let M_i be the number of paths of no more than i-hop from the source switch to an arbitrary switch. Since there are roughly $y+y(y-1)+...+y(y-1)^{i-1}$ such paths for each source switch, whose destinations are evenly distributed among N switches, the average M_i is

$$\overline{E}(M_i) = \frac{y+y(y-1)+...+y(y-1)^{i-1}}{N}.$$

In LLSKR, the "short" paths are no more than H-hop where H is the smallest integer such that

$$\overline{E}(M_H) = \frac{y+y(y-1)+...+y(y-1)^{H-1}}{N} > th_w,$$

where th_w is an parameter of LLSKR. In general, LLSKR has $1 \le th_w \le k$, where k is the number of paths for each SD pair. In our experiments, we set $th_w = 2$. The value of th_w determines the whole system path diversity of LLKSR: the th_w value decides H and $E(M_H)$, which is the expected number of paths between each pair of switches to route traffic for SD pairs on the pair of switches. For example, for $RRG(N = 2880, x = 48, y = 38)$ with $th_w = 2$, $H = 3$; and $\overline{E}(M_H) = \frac{38+38*37+38*37^2}{2880} = 18.6$. Using LLSKR, the vast majority of the traffic are carried over paths that are no more than H hops, for example 3 hops for $RRG(2880, 48, 38)$. Note that for $XGFT(3; 24, 24, 48; 1, 24, 24)$, the corresponding fat-tree of $RRG(2880, 48, 38)$, the paths between switches to carry traffic are mostly 4-hop paths.

For each pair of switches, LLSKR determines the M paths p_0, p_1, ..., p_{M-1} to be used for carrying traffic for SD pairs on the pair of switches as follows. LLSKR computes all paths that are H hops or less from the source switch to the destination switch. Let the number of such paths be X. If X is larger than or equal th_s where th_s is another algorithm parameter, LLSKR considers that the number of short paths between the pair of switch is sufficiently large, sets $M = \min\{X, k(x-y)^2\}$, and uses the M paths to carry traffic for the $(x-y)^2$ SD pairs on the pair of switches. The parameter th_s is the threshold determine whether there are a sufficient number of paths between one single pair of switches: if the number of paths between a pair of switches is too small, KSP will then be used to compute more paths for the pair. Thus, this threshold controls the trade-off between using a small number of short paths or using a larger number of longer paths. In our experiments, $th_s = 2$.

Since the topology is a random graph, sometimes $X < th_s$. In this case, LLSKR uses KSP to get k shortest paths between the two switches and use the k paths for the SD pairs: the paths selected are the same as KSP and some longer paths may be used to carry traffic. Note that th_w and th_s should be selected such that the chance for $X < th_s$ is small: the vast majority of traffic follows paths of no more than H hops between switches.

After LLSKR determines the M paths between a pair of switches, it uses the spread mechanism to allocate k paths for each pair of SD pairs on the pair of switches. Let us number of the $x-y$ nodes in the source switch to be s_0, ..., s_{x-y-1} and the $x-y$ nodes in the destination switch to be d_0, ..., d_{x-y-1}. There are several options to evenly distribute (spread) the M paths, p_0, p_1, ..., p_{M-1} between switches to the $(x-y)^2$ SD pairs. One option is to order the $(x-y)^2$ SD pairs in some way and use the modulo operation to determine the k paths for each SD pair. For example, the $(x-y)^2$ SD pairs can be ordered as: (s_0,d_0), (s_0,d_1), ..., (s_0,d_{x-y-1}), (s_1,d_0), (s_1,d_1), ..., (s_1,d_{x-y-1}), ..., (s_{x-y-1},d_0), (s_{x-y-1},d_1), ..., (s_{x-y-1},d_{x-y-1}). (s_0,d_0) will be routed using k paths p_0, p_1, ..., $p_{(k-1) \bmod M}$, (s_0,d_1) will be routed using k paths $p_{k \bmod M}$, $p_{(k+1) \bmod M}$, ..., $p_{(2k-1) \bmod M}$. In general, (s_i,d_j), $0 \le i,j \le x-y-1$, will be routed using paths $p_{((i\times k+j)\times k) \bmod M}$, $p_{((i\times k+j)\times k+1) \bmod M}$, ..., $p_{(i\times k+j+1)\times k-1 \bmod M}$. Using this option, the paths for a

SD pair depend both on the source node and the destination node. The paths may also be distributed such that the paths for each SD pair depend on only the source node or the destination node. For example, we can order the destination as d_0, d_1, ..., d_{x-y-1} and route the SD pairs as follows. SD pairs to d_0 ((s_0, d_0), (s_1, d_0), ..., (s_{x-y-1}, d_0)) use paths p_0, p_1, ..., $p_{(k-1) \bmod M}$; SD pairs to d_1 ((s_0, d_1), (s_1, d_1), ..., (s_{x-y-1}, d_1)) use paths $p_{k \bmod M}$, $p_{(k+1) \bmod M}$, ..., $p_{(2k-1) \bmod M}$, and so forth. In general, SD pairs to d_i, $0 \leq i \leq x-y-1$ uses paths $p_{i \times k \bmod M}$, $p_{(i \times k+1) \bmod M}$, ..., $p_{(i \times k+k-1) \bmod M}$. In our experiments, we use this destination-based path allocation.

LLSKR is described in Figure 2. The algorithm first computes all paths between switches that are no more than H hops and stores the paths in $P[ssw][dsw]$ (Lines (1) to (3)). $P[ssw][dsw][i]$ is the i-th path. After that, the algorithm checks if every pair of switches has a sufficient number (th_s) number of paths. If it does not, the k-shortest path routing algorithm is used to find k shortest paths for the pair of switches; and the paths are stored in $P[ssw][dsw]$. Once the paths are computed for each pair of switches, the M paths between each pair of switches are spread over the $(x-y)^2$ SD pairs on the pair of switches. In Figure 2, the paths are spread using the destination-based scheme discussed earlier (line (8)).

Input: jellyfish topology $RRG(N, x, y)$, th_w, th_s, source node src, destination node dst.
Output: k paths

(1) Let H be the small integer such that $\frac{y+y^2+...+y^H}{N} > th_w$.
(2) Compute all no more than H-hop paths between switches
(3) Let $P[ssw][dsw]$ be the set of paths from switch ssw to switch dsw: $P[ssw][dst][i]$ is the i-th path
(4) If $|P[ssw][dsw]| < th_s$, compute k shortest-paths from ssw to dsw and store in $P[ssw][dsw][..]$.
(5) Let l_s be the local number of src.
(6) Let l_d be the local number of dst.
(7) Let $M = |P[ssw][dsw]|$.
(8) The k paths from src to dst are: $P[ssw][dsw][l_d * k \% M]$, $P[ssw][dsw][(l_d * k + 1) \% M]$, ..., $P[ssw][dsw][(l_d * k + k - 1) \% M]$

Figure 2: LLSKR routing algorithm

LLSKR is able to exploit multi-path in a more flexible manner than KSP. By tuning th_w value, LLSKR can allow more paths between switches to be used while maintaining the average path lengths than KSP. Consider for example, routing on $RRG(2880, 48, 38)$ with 8-path routing ($k = 8$). Using KSP, only 8 paths will be used to carry traffic for all SD pairs on the pair of switches. Since the probability that there is a one-hop path between the switches is $\frac{38}{2880}$, the probability that there is a 2-hop path between the switches is $\frac{38 \times 37}{2880}$, the rest of the paths will be at least three hops. Hence, the average path length with KSP is at least

$$\frac{\frac{38}{2880} * 1 + \frac{38*37}{2880} * 2 + (8 - \frac{38}{2880} - \frac{38*37}{2880}) * 3}{8} = 2.94.$$

With LLSKR ($th_w = 2$, $th_s = 2$), we have $H = 3$ and $\overline{E}(M) = 18.6$. The average number of paths between switches to be used is 18.6, and the average path length is roughly

$$\frac{\frac{38}{2880} * 1 + \frac{38*37}{2880} * 2 + (18.6 - \frac{38}{2880} - \frac{38*37}{2880}) * 3}{18.6} = 2.98.$$

By tuning the value for th_s, LLSKR will use short paths for most of the SD pairs and use longer paths only when it is absolutely necessary (not enough short paths between two switches). Consider again $RRG(2880, 48, 38)$, since this is a random topology, the number of short (3-hop) paths between a pair of switch is a random number. Assume that for a pair of switches, the number of such paths is 5. Using KSP, 3 long paths will be used to route the traffic. In this case, the routing performance may be improved by enforcing that only the 5 short paths are used. LLSKR has the flexibility for different trade-offs to achieve the maximum performance.

LLSKR can be realized using source routing. However, both LLSKR and KSP are not destination based routing. They can be realized in the current InfiniBand networks, which mainly supports destination based routing, using the destination renaming technique [10] that requires multiple addresses to be associated with each node. This limits the scalability of such a system.

4.1 Performance of LLSKR

We compare LLSKR with KSP on a number of jellyfish topologies, including both large and small topologies constructed using 24-, 36-, and 48-port switches. The large topologies correspond to 3-level fat-trees with similar costs including a $RRG(2880, 48, 38)$ that supports 28800 processing nodes with 2880 48-port switches, a $RRG(1620, 36, 28)$ that supports 12960 processing nodes with 1620 36-port switches, and a $RRG(720, 24, 19)$ that supports 3600 processing nodes with 720 24-port switches. The small topologies correspond to 2-level fat-trees with similar costs including a $RRG(72, 48, 32)$ that supports 1152 processing nodes with 72 48-port switches, a $RRG(54, 36, 24)$ that supports 648 processing nodes with 54 36-port switches, and a $RRG(36, 24, 16)$ that supports 288 processing nodes with 36 24-port switches.

Since RRG's are random graphs that include some adversary topologies such as disconnected graph, our experimental software has a sanity check that throws away disconnected topologies. However, throughout all of our experiments, this sanity check was never triggered. We perform experiments with different random topologies with different random seeds, we found consistently that the performance trend for the same type of RRG's is virtually identical. For example, for 35 randomly generated RRG(72, 48, 32)'s with whole system random map with 8-path LLSKR, the overall average throughput across all patterns considered for the topologies ranges from 0.89409(89.409%) to 0.90305(90.305%) with an average of 0.89891 and a standard deviation of 0.00259. Virtually any RRG with the same parameters is representative of the performance characteristics for the type of RRGs. Hence, the results that we report is

an arbitrarily selected RRG (generated with random seed 1201).

Table 2 summarizes the topologies used in this subsection and lists the value of H and the expected number of paths between each pair of switches ($\overline{E}(M)$). We will use the notation KSP(a) for k-shortest path routing with $k = a$ and $LLSKR(a)$ for the proposed limited length spread k-shortest path routing with $k = a$. The algorithm parameters for LLSKR $th_w = 2$ and $th_s = 2$ for all the experiments.

topology	sw. size	# of sw.	# of nodes	H	$\overline{E}(M_H)$
$RRG(36, 24, 16)$	24	36	288	2	7.1
$RRG(54, 36, 24)$	36	54	648	2	10.7
$RRG(72, 48, 32)$	48	72	1152	2	14.2
$RRG(720, 24, 19)$	24	720	3600	3	9.1
$RRG(1620, 36, 28)$	36	1620	12960	3	13.1
$RRG(2880, 48, 38)$	48	2880	28800	3	18.6

Table 2: Jellyfish topologies studied

Figure 3 shows the performance of the HPC traffic patterns on $RRG(2880, 48, 38)$ with direct map assuming $k = 8$. For this topology, $H = 3$ and the expected number of paths per switch pair is 18.6. The performance is presented as the percentage of the injection rate, which is the flow rate when the network is one giant hyperthetical crossbar switch. For example, for 2DNN pattern with KSP(8), we have a performance of about 95%. This means that the average per flow throughput for all 2DNN patterns is 95% of the injection rate: the performance of the network is 95% of a crossbar switch with the same port bandwidth. As can be seen from the figure, $LLSKR(8)$ achieves better performance than $KSP(8)$ for all patterns with the largest improvement of 38.8% for the shift pattern. On average across all patterns, $LLSKR(8)$ performs better by 10.2%. Figure 4 shows the results on the same topology with random map. $LLSKR(8)$ is consistently better with the average improvement of 9.4% across all patterns. Similar trend is also observed for $RRG(1620, 28, 20)$ and $RRG(720, 24, 19)$: $LLSKR(8)$ consistently achieves higher performance than $KSP(8)$ across all of the traffic patterns on all of the topologies, which demonstrates that LLSKR is a more robust and efficient routing scheme for jellyfish. Since LLSKR consistently out-performs KSP with the same k for all patterns, we will only report the average of all patterns in the rest of the section with each type of patterns carries the same weight.

Figures 5 and 6 compare the average performance across all patterns on small and large topologies built by different switches. Figures 5 shows the results with whole system direct map on small topologies, $RRG(36, 24, 16)$, $RRG(54, 36, 24)$, and $RRG(72, 48, 32)$. For all of these topologies, $H = 2$ and the average number of paths per pair of switches is 7.1 for $RRG(36, 24, 16)$, 10.7 for $RRG(54, 36, 24)$, and 14.2 for $RRG(72, 48, 32)$. As can be seen from the figure, even for $RRG(36, 24, 16)$ with a 7.1 expected number of paths, LLSKR(8) is still better than KSP(8). There are several factors contributing to this. First, 7.1 is the expected number of paths, there are pairs of switches with more short paths that can be exploited by LLSKR(8). Second, in case when the number of paths

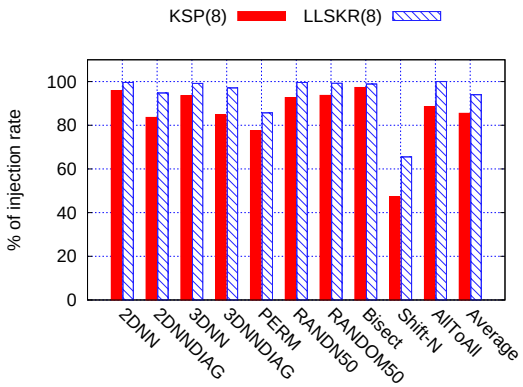

Figure 3: Performance of different patterns on RRG(2880, 48, 38) with whole system direct map ($k = 8$)

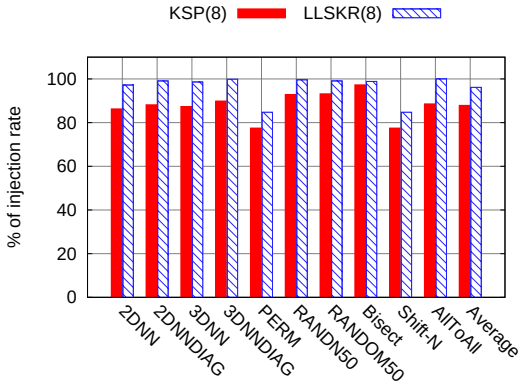

Figure 4: Performance of different patterns on RRG(2880, 48, 38) with whole system random map ($k = 8$)

(M) between a switch pair is less than 8, but larger than $th_s = 2$, LLSKR only uses the short paths while KSP(8) use some $(8 - M)$ longer paths. Larger improvement is observed when the expected number of paths is significantly larger than k (RRG(72, 48, 32)) when LLSKR exploits more path diversity. Figures 6 shows the average performance across all patterns with whole system random map on large topologies, $RRG(720, 24, 19)$, $RRG(1620, 36, 28)$, and $RRG(2880, 48, 38)$. For all of these topologies, $H = 3$ and the average number of paths per pair of switches is 9.1 for $RRG(720, 24, 19)$, 13.1 for $RRG(1620, 36, 28)$, and 18.6 for $RRG(2880, 48, 38)$. The trend is similar to the small topology experiments: LLSKR(8) is consistently better with larger improvement observed when the expected number of paths is larger (RRG(2880, 48, 38)).

Figure 7 shows the performance improvement percentage of LLSKR over KSP for different k on $RRG(1620, 36, 28)$ with whole system direct and random mapping. For the case where k values are 1 and 2, LLSKR improves the overall performance by 28.9% and 30.1% for direct mapping respectively, and in the case of random mapping 14.3% and 24.1%, respectively. For larger k values of 4 and 8, the performance gain starts to diminish, with $k = 8$ improving approximately 5% for both direct and random mapping. For smaller k's the $(x - y)^2$ SD pairs share k paths when using KSP where LL-

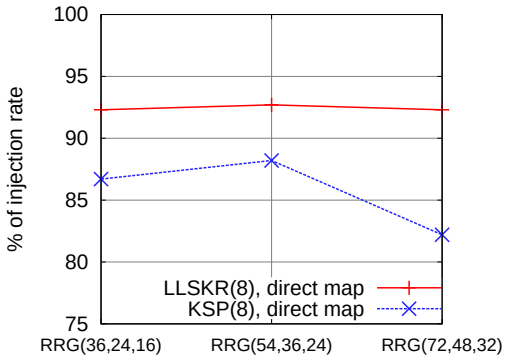

Figure 5: Performance on small networks with different switch sizes

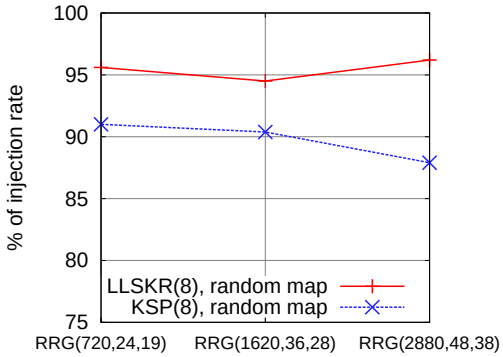

Figure 6: performance on large networks with different switch sizes

SKR exploits the path diversity more effectively. As the value of k increases, $KSP(k)$ becomes more effective in exploiting path diversity. This explains the narrower gap between LLSKR and KSP as k increases. Nonetheless, the advantage of LLSKR is consistent and noticeable.

In summary, the results of our experiments demonstrate that LLSKR is a more efficient and robust routing scheme for the jellyfish topology than KSP by more effectively exploiting the path diversity. It consistently results in higher average throughputs than KSP across all traffic patterns we consider with up to 30% performance improvement when k is small.

5. RELATIVE PERFORMANCE OF FAT-TREES AND JELLYFISH WITH HPC WORKLOADS

We compare the performance of jellyfish with fat-trees with HPC workloads in this section. Like in [14], we focus on comparing the performance of full bisection bandwidth fat-trees with corresponding jellyfish topologies. Full bisection bandwidth fat-trees have constraints on the number of switches and the number of processing nodes in the tree while the jellyfish topologies can support any number of switches, but have constraint on the number of processing nodes since all switches must have the same number of processing nodes. To compare fat-tree with jellyfish with similar

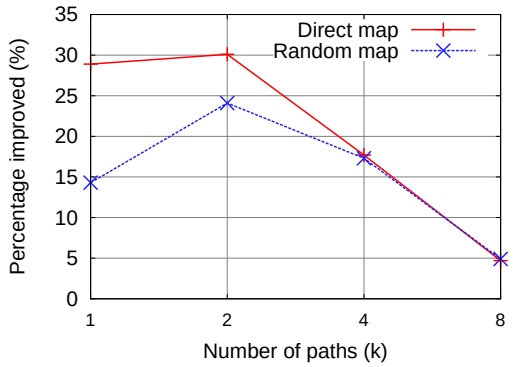

Figure 7: Improvement percentage of LLSKR over KSP for different k on $RRG(1620, 36, 28)$

costs, we fix fat-tree topologies and compare the fat-trees to corresponding jellyfish topologies with the same switch size and same number of switches, and a similar number (equal or slightly larger) of processing nodes. More specifically, let N be the number of switches, np_{ftree} be the number of processing nodes in a fat-tree. The number of ports connecting to processing nodes in the corresponding jellyfish is $\lceil \frac{np_{tree}}{N} \rceil$ and the total number of processing nodes in the jellyfish is thus $\lceil \frac{np_{tree}}{N} \rceil \times N$. In Table 3, we show topologies for switch sizes 24, 36, and 48 for 2-level and 3-level trees, and their corresponding jellyfish topologies (other parameters for the jellyfish topologies are given in Table 2). We consider multi-path routing with a MPTCP-like protocol for both topologies. The multi-path routing on fat-tree uses the disjoint path heuristic [11] that has higher performance than other path allocation scheme. When $k = 1$, the disjoint path heuristic is equivalent to Destination-mod-k routing. The multi-path routing scheme for jellyfish is LLSKR.

fat-tree topology	sw. size	# of sw.	# of nodes	jellyfish topology
$(2; 12, 24; 1, 12)$	24	36	288	$RRG(36, 24, 16)$
$(2; 18, 36; 1, 18)$	36	54	648	$RRG(54, 36, 24)$
$(2; 24, 48; 1, 24)$	48	72	1152	$RRG(72, 48, 32)$
$(3; 12, 12, 24; 1, 12, 12)$	24	720	3456	$RRG(720, 24, 19)$
$(3; 18, 18, 36; 1, 18, 18)$	36	1620	11664	$RRG(1620, 36, 28)$
$(3; 24, 24, 48; 1, 24, 24)$	48	2880	27648	$RRG(2880, 48, 38)$

Table 3: Fat-trees and the corresponding jellyfish topologies used in the study

Figure 8 compares the performance of $RRG(36, 24, 16)$ and $XGFT(2; 12, 24; 1, 12)$ with the whole system direct map and $k = 8$. Figure 9 shows the results with the whole system random map. In both cases, the fat-tree offers higher performance across all traffic patterns. Same observation is made for other small topologies, different values of k, and different node mapping schemes. For all experimental configurations that we perform, 2-level fat-trees consistently offer higher average performance than the corresponding jellyfish for all HPC traffic patterns considered. Thus, we conclude that for small networks that correspond to 2-level fat-trees, fat-tree topologies are more effective than the jellyfish topology.

Figure 10 compares the performance of $RRG(2880, 48, 38)$ and $XGFT(3; 24, 24, 48; 1, 24, 24)$ with the whole system di-

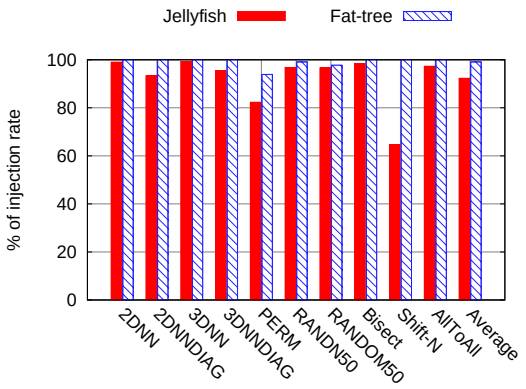

Figure 8: Performance of different traffic patterns for $XGFT(2;12,24;1,12)$ and $RRG(36,24,16)$ with whole system direct map

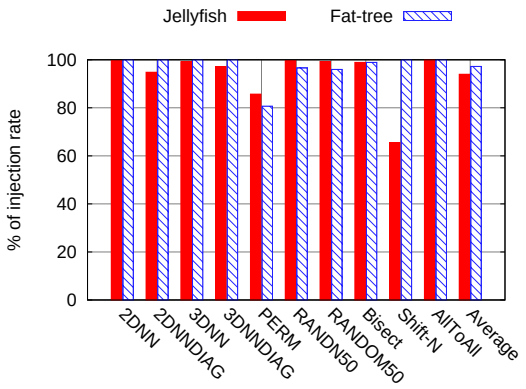

Figure 10: Performance of different traffic patterns for $XGFT(3;12,12,24;1,12,12)$ and $RRG(720,24,19)$ with whole system direct map, $k=8$

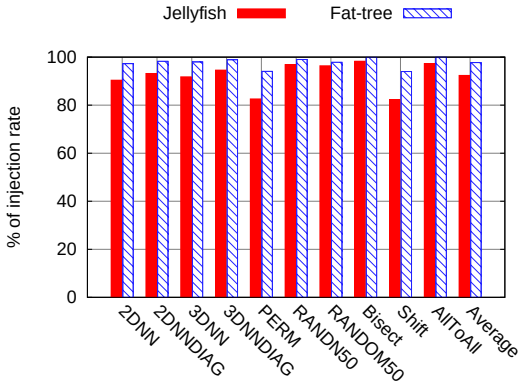

Figure 9: Performance of different traffic patterns for $XGFT(2;12,24;1,12)$ and $RRG(36,24,16)$ with whole system random map

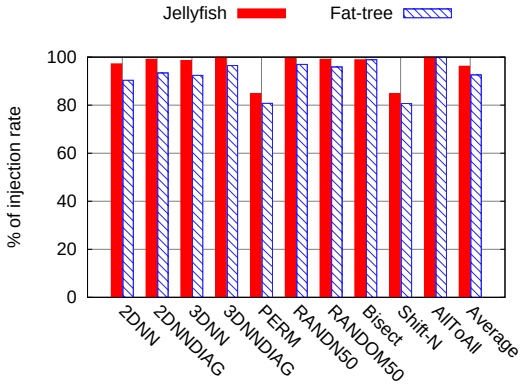

Figure 11: Performance of different traffic patterns for $XGFT(3;12,12,24;1,12,12)$ and $RRG(720,24,19)$ with whole system random map, $k=8$

rect map and $k=8$. Figure 11 shows the results with the whole system random map. Using direct map, the fat-tree offers high performance for stencil and shift patterns, while jellyfish offer higher performance for other global traffic patterns such as permutation, RANDOM50 and RANDN50 patterns. This confirms the results in [14] that some jellyfish topologies have larger bisection bandwidth and higher permutation throughput than the corresponding fat-trees. Across all patterns, the fat-tree is better by 3.4%. With whole system random mapping in Figure 11, jellyfish is consistently better for all patterns except bisect and alltoall patterns when both jellyfish and fat-tree perform similarly. Across all patterns, with random mapping, the jellyfish topology is 3.9% better. For both random and direct mapping, both topologies achieve more than 90% of the throughput: both are quite effective for HPC traffic. However, jellyfish is better in dealing with random and global traffic while fat-tree is better with local stencil traffic patterns produced from direct mapping.

Figure 12 shows the average throughput across all traffic patterns for 3-level fat-trees with different switch sizes and their corresponding jellyfish topologies. The results are consistent: fat-trees are slightly better for direct map while jellyfish is slightly better for random map for all different switch sizes. In general, jellyfish offer higher throughput for random global traffic patterns while the fat-tree supports HPC patterns with direct map better. In practice, traffic patterns are not generated from whole system direct map or random map. In most of practical cases, a job will get multiple contiguous blocks of processing nodes in the system. The job allocation trace-based mapping reflects this realistic situation. Figure 13 shows the average throughput for the trace-based mapping. Not surprisingly, the performance difference between the two topologies for this mapping scheme is very small. We conclude that both topologies achieve similar performance using the trace-based mapping scheme. Notice that the absolute throughput is slightly larger using the trace-based mapping scheme. This is because using this mapping, the traffic pattern is only generated for the largest job: the network is not as loaded as when the whole system is running a job.

Figure 14 and Figure 15 show the performance scalability of fat-tree and jellyfish. The average throughput for 4-path and 8-path routing across all traffic patterns that we consider for 2-level, 3-level, and 4-level fat-trees with 24-port switches and the corresponding jellyfish topologies are given in Figure 14. This experiment uses the trace-based mapping. The results shows that an opposite trend for fat-tree and jellyfish: as the tree height increases, the performance of fat-tree decreases while the performance for the corre-

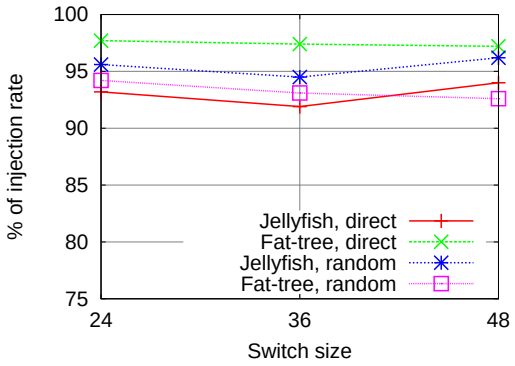

Figure 12: Average performance for 3-level trees and their corresponding jellyfish topologies with different switch sizes, $k = 8$

sponding jellyfish increases. This indicates that jellyfish is more scalable than fat-tree. We note that the performance changes are relatively small in this case. This is because the results are the average of many types of traffic patterns, and becuase both jellyfish and fat-tree achieve similar high performance (close to 100% injection rate) for many types of patterns. Figure 15 shows the average throughput for permutation communication only. As can be seen from the figure, for the permutation communication, the performance decrease of the fat-tree and the performance increase of jellyfish as the network becomes larger are very significant.

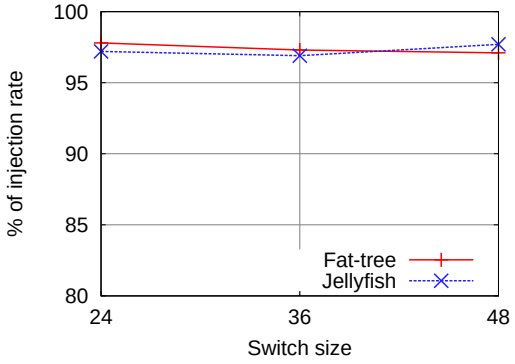

Figure 13: Average performance for 3-level trees and their corresponding jellyfish topologies with different switch sizes with trace-based process mapping

6. CONCLUSIONS

We propose a new routing scheme LLSKR for jellyfish. LLSKR exploits the path diversity in jellyfish more effectively than the existing k-shortest path routing, and consistently achieves higher performance for different network configurations and traffic patterns. The improvement is particularly large when the number of paths for each SD pair is limited to a small number. We comprehensively compare the performance of jellyfish and fat-tree using HPC workloads. The results indicate that for HPC workloads, 3-level fat-trees offer similar performance to the corresponding jellyfish topologies with similar costs. Fat-trees achieve higher

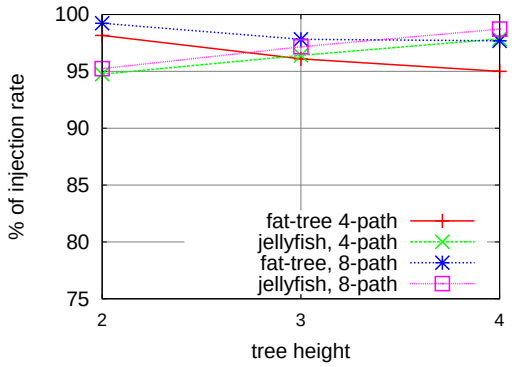

Figure 14: Average performance for fat-trees of different levels with 24-port and their corresponding jellyfish topologies with trace-based mapping

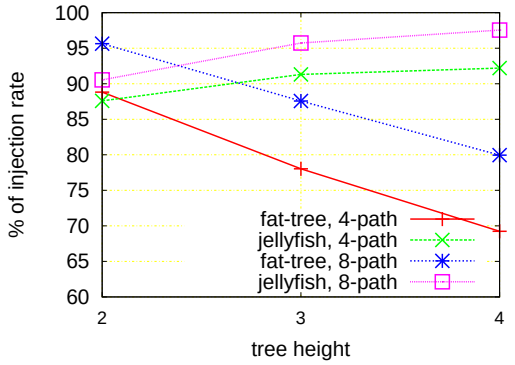

Figure 15: Average permutation performance for fat-trees of different levels with 24-port and their corresponding jellyfish topologies with trace-based mapping

performance for smaller systems that can be built with 2-level fat-trees while jellyfish has better scalability.

Acknowledgments

This work was performed at the Ultrascale Systems Research Center (USRC) at Los Alamos National Laboratory, supported by the U.S. Department of Energy DE-FC02-06ER25750. The publication has been assigned the LANL identifier LA-UR-13-22450.

7. REFERENCES

[1] D. Bertsekas and R. Gallager, *Data Networks*, Englewood Cliffs, NJ:Prentice-Hall, 1992.

[2] B. Bollobas and W. F. de la Vega, "The Diameter of Random Regular Graphs," *Combinatorica*, 2(2)125-134, 1982.

[3] B. Bollobas, "The Isoperimetric Number of Random Regular Graphs," *European Journal of Combinatorics*, 9:241-244, 1988.

[4] William James Dally and Brian Patrick Towles, *Principles and Practices of Interconnection Networks*, Morgan Kaufmann, 2003.

[5] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan kaufmann, 2003.

[6] H. Han, S. Shakkottai, C. V. Hollot, r. Srikant, and D. Towsley, "Multi-path TCP: a Joint Congestion Control and Routing Scheme to Exploit Path Diversity in the Internet," *IEEE/ACM Transactions on Networking*, 14(6), Dec. 2006.

[7] T. Hoefler, T, Schneider, and A. Lumsdaine, "Multistage Switches are not Crossbars: Effects of Static Routing in High-Performance Networks," *IEEE International Conference on Cluster Computing*, pages 116-125, 2008.

[8] C. Hopps, "Analysis of an Equal-Cost Multi-Path Algorithm," *RFC 2992*, November 2000.

[9] D. J. Kerbyson, K. J. Barker, "Automatic Identification of Application Communication Patterns via Templates," *18th International Conf. on Parallel and Distributed Computing Systems*, September, 2005.

[10] P. Lopez, J, Flich, and J, Duato, "Deadlock-free Routing in InfiniBand through Destination Renaming," *International Conference on Parallel Processing* (ICPP), pages 427-434, Sept. 2001.

[11] S. Mahapatr, X. Yuan and W. Nienaber, "Limited Multi-path Routing on Extended Generalized Fat-trees," the *2nd IPDPS Workshop on Communication Architecture for Scalable Systems* (CASS'12), May 2012.

[12] S. R. Ohring, M. Ibel, S. K. Das, and M. Kumar, "On Generalized Fat Trees," *Proceedings of the 9th International Parallel Processing Symposium*, pages 37-44, 1995.

[13] P.G. Raponi, F. Petrini, R. Walkup, and F. Checconi, "Characterization of the Communication Patterns of Scientific Applications on Blue Gene/P", *2011 IEEE IPDPS Workshops and PhD Forum*, pages 1017-1024, 2011.

[14] A. Singla, C-Y. Hong, L. Popa, and P. B. Godfrey, "Jellyfish: Networking Data Centers Randomly," *9th USENIX Symposium on Networked Systems Design and Implementation* (NSDI), April 2012.

[15] D. Wischik, C. Raiciu, A. Greenhalgh, and M. Hanley, "Design, Implementation, and Evaluation of Congestion Control for Multi-path TCP," *8th USENIX Symposium on Networked Systems Design and Implementation* (NSDI), April 2011.